I. PROCESSES

The Process Abstraction

computers do „several things at the same time“ (just looks this way though quick process switching (**Multiprogramming**))

⇝ **process** abstraction models this concurrency:
- container contains information about program execution
- conceptually, every progress has own „virtual CPU“
- execution context is changed on process switch
- dispatcher switches context when switching processes
- **context switch**: dispatcher saves current registers/memory mappings, restores those of next process

Process-Cooking Analogon

Program/Process like Recipe/Cooking

Recipe: lists ingredients, gives algorithm what to do when
⇝ program describes memory layout/CPU instructions

Cooking: activity of using the recipe
⇝ process is activity of executing a program

multiple similar recipes for same dish
⇝ multiple programs may solve same problem

recipe can be cooked in different kitchens at the same time
⇝ program can be run on different CPUs at the same time (as different processes)

multiple people can cook one recipe
⇝ one process can have several worker threads

Concurrency vs. Parallelism

OS uses currency + parallelism to implement multiprogramming

1. **Concurrency**: multiple processes, one CPU
   ⇝ not at the same time
2. **Paralellism**: multiple processes, multiple CPU
   ⇝ at the same time

Virtual Memory Abstraction – Address Spaces

every process has own *virtual addresess* (vaddr)

MMU relocates each load/store to *physical memory* (pmem)

processes never see physical memory, can't access it directly

\+ MMU can enforce protection (mappings in kernel mode)

\+ programs can see more memory than available
80:20 rule: 80% of process memory idle, 20% active
can keep working set in RAM, rest on disk

– need special MMU hardware

Address Space (Process View)

code/data/state need to be organized within process
⇝ **address space layout**

Data types:

1. *fixed size* data items
2. data naturally *free'd in reverse allocation order*
3. data *allocated/free'd „randomly“*

compiler/architecture determine how large int is and what instructions are used in text section (`code`)

Loader determines based on exe file how executed program is placed in memory

Segments – Fixed-Size Data + Code

some data in programs never changes or will be written but never grows/shrinks
⇝ memory can be statically allocated on process creation

BSS segment (*block started by symbol*):
- statically allocated variables/non-initialized variables
- executable file typically contains starting address + size of BSS
- entire segment initially 0

Data segment:
- fixed-size, initlized data elements (e.g. global variables)

read-only data segment:
- constant numbers, strings

All three sometinmes summarized as one segment

compiler and OS decide ultimately where to place which data/how many segments exist

Segments – Stack

some data naturally free'd in reverse allocation order
⇝ very easy memory management (stack grows upwards)

fixed segment starting point

store top of latest allocation in **stack pointer** (SP) (initialized to starting point)

allocate `a` byte data structure: `SP += a; return(SP - a)`

free `a` byte data structure: `SP -= a`

Segments – Heap (Dynamic Memory Allocation)

some data „randomly“ allocated/free'd

two-tier memory allocation:

1. allocate large memory chunk (**heap segment**) from OS
   - base address + **break pointer** (BRK)
   - process can get more/give back memory from/to OS

2. dynamically partition chunk into smaller allocations
   - `malloc`/`free` can be used in random oder
   - purely user-space, no need to contact kernel